L14: Entry 31 of 33 File: USPT Oct 19, 1999

DOCUMENT-IDENTIFIER: US 5970490 A
TITLE: Integration platform for heterogeneous databases

Abstract Text (1):
A method for processing heterogeneous data including high level specifications to drive program generation of information mediators, inclusion of structured file formats (also referred to as data interface languages) in a uniform manner with heterogeneous database schema, development of a uniform data description language across a wide range of data schemas and structured formats, and use of annotations to separate out from such specifications the heterogeneity and differences that heretofore have led to costly special purpose interfaces with emphasis on self-description of information mediators and other software modules.

Brief Summary Text (2):
Currently, databases used for design and engineering employ a variety of different data models, interface languages, naming conventions, data semantics, schemas, and data representations. Thus a fundamental problem for concurrent engineering is the sharing of heterogeneous information among a variety of design resources. Successful concurrent engineering also requires access to data from multiple stages of the design life-cycle, but the diversity among data from different tools and at different stages creates serious barriers.

Brief Summary Text (3):
Although there have been several efforts directed at heterogeneous databases, a significant need continues to exist for design, engineering, and manufacturing applications to be able to easily access and import heterogeneous data. Efforts to develop global query languages do not address the large group of users who want to see the world of external data as if it were an extension of their existing system and its specialized representation. These users may not wish to learn a different global representation--and more importantly, their expensive design tools can only work with their one specialized representation.

Brief Summary Text (4):
Database gateways, Common Object Request Broker (CORBA), and Open Database Connectivity (ODBC) interfaces which purport to address heterogeneity only do so at a relatively superficial level. Database gateways provide communication mechanisms to external systems--but only if those systems have a relational interface. That is, the system must either processes SQL queries or at least must provide a relational application programming interface (API). Since an API is a set of functions to be accessed via remote or local procedure invocation, a relational API presents data in terms of named arrays (relation tables) that contain flat tuples, with one data value per column in each tuple, and with the number of columns fixed per named array. The functions' names and parameters are defined in the API and may vary depending upon the gateway. Application programmers must write their programs to invoke the functions of a particular API. The proposal for an ODBC standard offers the potential for a single agreed upon naming of the functions in the API. In addition, the CORBA approach allows more than a single API `standard` to coexist, as it can locate an appropriate interface from a library of interfaces.

Brief Summary Text (7):

Previous related work in the database field can be grouped roughly into three areas: access to heterogeneous databases (HDBs), schema integration, and object encapsulation of existing systems.

Brief Summary Text (8):
Access to HDBs: Language features for multidatabase interoperability include variables which range over both data and metadata, including relation and database names, and expanded view definitions with provisions for updatability. Selected translation or mapping approaches are known and others which are based upon the 5-level architecture are described in Sheth and Larson. Approaches to semantics-based integration and access to heterogeneous DBs where semantic features are used to interrelate disparate data and resolve potentially different meanings are known. Seen from a higher level, what are needed are mediation services among different representations and systems.

Brief Summary Text (9):
Schema Integration: Database design tools have been applied to schema integration, and related work formalizes interdatabase dependencies and schema merging. Some related approaches utilize view-mapping, and data migration. Some of the work in this area assumes that the resulting target schema or view is to be relational rather than another heterogeneous representation. Much of the work on data semantics also is applicable to semantics of schema integration.

Brief Summary Text (10):
Object Encapsulation: Since object technology can hide the implementation and data structures within objects, it is interesting to consider encapsulation and the hiding of heterogeneity within object interfaces, with one or more interfaces being specially crafted for each heterogeneous database. Some of the work in this area includes the FINDIT system [6], object program language interfaces, and an execution environment for heterogeneous software interfaces. These object approaches serve to hide, rather than obviate, the specialized programming which still is needed for each application. Sciore has done interesting work on the use of annotations to support versioning, constraint checking, defaults, and triggers.

Brief Summary Text (13):
The DAtabase Integration SYstem (DAISy) of the present invention provides both high-level user interfaces and program level access across heterogeneous databases (HDBs), allowing integration of a wide variety of information resources including relational and object databases, CAD design tools, simulation packages, data analysis and visualization tools, and other software modules. DAISy eliminates tedious and costly specialized translators. A key focus is on reusability of components through a specification-based approach. A declarative specification language is utilized to represent the source and target data representations. This high level data structure specification (HLDSS) provides a uniform language for representing diverse databases and specialized file formats, such as produced by CAD tools. In addition, a rule-like specification applies functional transformations in a manner similar to that of production rule systems and term-rewriting systems. These specifications are utilized to create information mediators and information bridges each of which can access heterogeneous data resources and transform that information for use by databases and specialized representations. In all cases, the data is mapped to an intermediate internal format. This also provides reusability of each information mediator to support multiple applications. Each of the information bridges are created through a program generation process, and this process and the associated tools within the DAISy platform are reused for each new information bridge which is created.

Drawing Description Text (3):
FIG. 1 shows the architecture of the database integration platform;

Detailed Description Text (5):

When a standardized model or other neutral model is selected, then for N client sites or databases, one needs 2N transformations--i.e. N bi-directional mappings between each client information resource and the central model. Creation of such standardized application models is the goal of various standards efforts, including the PDES/STEP effort--DAISy can interface with such standardized models. When a standardized model is not available, other approaches may be pursued.

Detailed Description Text (14):
In this manner, a variety of kinds of information resources utilizing a common specification language and internal representation can be supported. The design can support heterogeneous databases with different schemas and data models, such as multiple relational and object-oriented databases, and other application packages, such as CAD, CAE, and CASE (Computer Aided Design, Computer Aided Engineering, and Computer Aided Software Engineering) tools, which produce specialized data structure representations.

Detailed Description Text (15):
The overall architecture of the system is reviewed in FIG. 1 and consists of several primary modules. The information bridge 1 transforms data from heterogeneous data resources 2, 4 and 6, for example, respectively a Nano-Fabrication Database, Simulation Tools, and a CAD/CAM (Computer Aided Design/Computer Aided Manufacturing) Database into a common intermediate representation and then into a specialized target representation--as determined by the specifications. The target representation may be another database with a different data model and schema, or the target may be a specialized data structure needed by a design tool.

Detailed Description Text (23):
A new software package then can be registered automatically once the existence of the package is known. In this case, the server invokes the package with the standard argument requesting its self-description, and the server then enters that information into its database. Later if the server is queried as to the existence of a software package that provides some set of functions, the server can respond by checking the attributes and values of registered packages, and return information on all that match the query. Other information can be provided in the self-description, such as the units of measurement that are used, semantic details, general information, and perhaps cross-referencing of other documentation (e.g., manual pages or documents that describe some standard implemented by the software).

Detailed Description Text (26):
The two HLDSS data structure specifications describe the source 22 and target 24 representations respectively. Both the source 22 and target 24 may be very different heterogeneous data resources. Each may be an object, relational, or other database, or a specialized data file and structure, or an application program interface (API) to a software package, such as a CAD tool.

Detailed Description Text (34):
The annotations/attributes are processed in both cases to obtain information such as database and API interfaces, hosts and filenames. The data handling information is stored on the individual nodes of the schema trees and code is generated to access input data and produce output data. The stored information differs between the source and target, so as to distinguish between input schema trees and output schema trees. However, the two generators are logically similar and may be combined.

Detailed Description Text (39):
Turning now to a more detailed description of the IA shown in FIG. 2, the high level data structure specification (HLDSS) consists of three parts and is used to describe the logical and physical structure of the source and target databases,

data structures, file formats, and associated information. Separate HLDSS's are used for the different source 22 (input) and target 24 (output) data. If a data representation is used for input in some cases and output in other cases, then essentially the same HLDSS can be used for both.

Detailed Description Text (40):
The HLDSS specification language is uniform regardless of the diverse database, data structure, and file representations. The significant differences among these different data representations is accounted for by annotations on the HLDSS productions (statements). These annotations give rise to different specialized interpretations of each production, based upon the annotation. The uniformity of the HLDSS language foreshadows the uniformity of the internal intermediate representation of the actual data within the information mediator/bridge 60.

Detailed Description Text (42):
Grammar productions specify the structure of the database, file format, or main memory data structures via a grammar-like set of extended BNF productions plus annotations. Each grammar production consists of:

Detailed Description Text (45):
The interpretation of these productions depends upon the optional annotations. If the annotation specifies "RDB:<database.handle>" then the production refers to a relational database table where the left hand side (LHS) of the production, e.g., \`Header\`, is the table name, and the right hand side (RHS) components are the attributes/columns which are to be selected. Each application of this production to the input would produce one such tuple from the table. The annotation may optionally specify a full SQL query, and this production will then refer to the resulting virtual table.

Detailed Description Text (46):
If the annotation is "ODB:<database.handle>" then the production refers to an object from an object-oriented database. The LHS is the name of the object, and the RHS components are the attributes or methods whose values are to be selected. An explicit OSQL query may be specified instead, in which case, the result will be treated as a virtual object (LHS) having the component names designated by the RHS.

Detailed Description Text (47):
The key unifying idea here is that all databases and data representations revolve around several kinds of named aggregates of subcomponents. In turn, each aggregate may participate in one or more higher level aggregates.

Detailed Description Text (54):
In all these cases, the approach uniformly treats the right hand side (RHS) of each production or statement as the components of the aggregate, and the name of the aggregate is given by the left hand side (LHS) of that production. Each instantiation of a production is treated as an instance of that kind of data aggregation. So for a relational database, the LHS names the relation (real or virtual/derived) and the RHS names the attribute fields. Then an instance of this production represents a relational data tuple.

Detailed Description Text (72):
This specification accesses a Sybase relational table named \`header\` for attributes Name, Version, Type, and Fmt. It also accesses an ONTOS object database to obtain the \`bound\` information from the \`Mesh\` object whose Name was just obtained from the relational tuple. The \`element\` and \`node\` data are obtained from the same Mesh object. \`numberofElements\` is the number of instances of \`element\` data that occur for \`element+\` and \`numberofNodes\` is the number of instances for \`node+\`. In turn, each \`element\` has an elementNumber and eight nodeNumber data components, each of which is an integer.

Detailed Description Text (73):
On lines which start with "Spec <name>:" there follows additional information about the annotation. This consists of a sequence of "Keyword: value" entries, where for certain keywords the value may be optional. The set of allowed keywords depends upon the main category or phyla of the data resource. For example, the annotation "{RDB: Syb}" applies to the production whose left hand side is "header". RDB signifies the relational family of database types, and "Syb" names the annotation. The elaboration of "Spec: Syb" here identifies the relational DB as being Sybase, with the server at "@site1.com", and here port 42 is indicated, though usually the port is not needed.

Detailed Description Text (74):
Similarly, if the annotation is "ODB:<tag>" then the production refers to an object from an object-oriented database. The LHS is the name of the object, and the RHS components are the attributes or methods whose values are to be selected. An explicit OSQL query may be specified to qualify the objects of interest, and/or to limit the set of attributes/methods to be accessed--not all OODBs support such queries.

Detailed Description Text (87):
1) The descriptive representation language or formalism is the logical structure diagram. It is a logical graph (often a tree) of data entities and logical dependencies. The fact that this descriptive language is uniform in its logical structure depends on the ability to independently specify the interpretation and naming relative to different data models and different databases.

Detailed Description Text (100):
The header tuple 120 represents data from a relational table while the data associated with the element object 140 comes from an object database--note that element represents a set or collection of element object instances. The sources of these and other data instances is described in the HLDSS specification.

Detailed Description Text (103):
Pattern matching can serve as a general paradigm for database access against object-oriented schemas, other data models, as well as for specialized design file structures. The approach is motivated by the observation that a query could be specified by a subgraph or subset of the database schema together with annotations designating the outputs and the selection predicates on certain components.

Detailed Description Text (104):
Database patterns can be seen as consisting of three aspects: (1) general pattern constructs, (2) data model specific constructs, and (3) reference to named components of a specific schema. As a result, the database patterns are applicable to object-oriented schemas and relational schemas as well as other data models. Queries may reference retrieval methods as well as explicitly stored attributes.

Detailed Description Text (105):
As a result, heterogeneous databases can be supported with capabilities such as: queries against data from different schemas, providing a uniform view over multiple databases, and providing different views for different users--e.g., with respect to a view commensurate with their local database.

Detailed Description Text (106):
A database pattern (DB pattern) may be defined in terms of selected components from the schema together with selection predicates. For an object-oriented database, the DB pattern will consist of objects, attributes and/or relationships from the schema, together with predicate qualifications to restrict the allowed data values and relationships. The restrictions may limit the combinations of data object instances which can successfully match the database pattern. Optional linking

variables can be utilized to interrelate several data objects and their attributes.

Detailed Description Text (107):
When the DB pattern is applied against a database of instances, components in the pattern match or bind to data instances such that all conditions in the pattern are satisfied. The attribute and object names utilized in the pattern serve as implicit variables which are bound to these names.

Detailed Description Text (109):
Database patterns provide the equivalents of selection, projection, and join. In addition they can provide recursion to obtain the closure of a relationship, as well as other constructs which are useful in non-relational schemas. The form of the patterns are based upon the structure of the schema, and thus such patterns are applicable not only to object oriented models but also to the relational model as well as more traditional hierarchical and network schemas.

Detailed Description Text (110):
A database pattern P is the ordered triple [ND,E,F], where ND is a set of node descriptors, each composed of a node N.sub.i and an optional predicate P.sub.i. The collection of individual nodes Ni constitutes the set N. E is a set of edges defined in terms of nodes from N, as discussed below. F is the pattern's outform, which provides a generalization of the notion of projection. F can be utilized to specify structuring and organization of the resultant data instances.

Detailed Description Text (111):
An edge E.sub.i represents a relationship among nodes. In the general case, an edge is indicated by a tuple (N1, N2, . . . , .theta., . . . , N.sub.K-1, N.sub.K) of nodes N.sub.i .di-elect cons. N, and thus may be a hyperedge. .theta. is an optional edge designator, which either may be an existing edge name, recursion over a given edge type, or a binary comparison function, such as a join condition. This general form of database pattern can be usefully treated as a hypergraph, in which some or all edges have k>2 nodes. When binary edges which consist of two nodes can be used, the database pattern P forms a graph.

Detailed Description Text (112):
The process of matching the pattern P against the database D treats the DB pattern P as a mapping (multi-valued function) which is applied to D to produce the match set M. This pattern matching process is denoted M=P(D). It produces a match set M which consists of all distinct outform/output instances.

Detailed Description Text (113):
The basic definition of database patterns is independent of the data model and the schema because the distinction is made between the following (conceptual) phases in the creation and use of database patterns. These are:

Detailed Description Text (114):
1. Syntax and structure of database patterns, as defined above, which are independent of data model.

Detailed Description Text (115):
2. Interpretations of database patterns relative to different data model formalisms: object-oriented, relational, etc. This impacts the meaning or interpretation of nodes and edges.

Detailed Description Text (116):
3. Labeling of a specific database pattern relative to a given application schema--including reference to attribute, relationship, and object names in the application schema. This impacts the naming of the nodes and edges of the DB Pattern.

Detailed Description Text (117):
4. Application of the pattern, including binding of pattern variables based on matches, and return of matching instances from the database.

Detailed Description Text (118):
These database patterns provide a graphical representation for global queries, and thereby support transparency with regard to location and heterogeneity of distributed data.

Detailed Description Text (153):
This comparison operation treats complex attributes (i.e. those with substructure) in terms of structure identity as the basis of equality matching. In effect this means that pointers may be compared without recursively descending the substructures. The implementation utilizes a hash join approach where a hash index is created for the tuples from the left relation based on its join key values, thereby reducing the potentially quadratic nature of a loop join into a process which is linear in the number of tuples.

Detailed Description Text (185):
The input tree 210 actually consists of both a schema tree and an instance tree. The input schema tree is constructed automatically from the input HLDSS and includes accessor functions which will retrieve and, if necessary, parse input data from multiple input sources. A node 212 in the schema tree represents a named type of data element, much as an attribute in a relational schema defines a component of each tuple. There also may be specific aggregation node(s) in a schema tree which represent a set, collection, or aggregation of subcomponents. Thus a roster may represent the set of students taking a course, or a database relation name may represent the set of tuples which populate that relation table.

Detailed Description Text (186):
When the accessor functions retrieve data instances, they build an instance tree which has the same logical structure as the schema tree, but now with potentially multiple instances nodes for those schema nodes which are subordinate to an aggregation node in the tree. There would be one input instance aggregate node for each occurrence of a collection--say each course--and one data node for each member of an aggregate (e.g. each student). Similarly, to represent a relation in a relational database, there would be a separate schema node for the relation itself, for a generic tuple, and for each attribute type. In the instance tree there would be as many tuple nodes as there are tuples, and each tuple would have one instance node for the value of each attribute of the relation

Detailed Description Text (215):
The type specifications, specifically integer, float and string, describe the data types of the actual instance data. The annotations and spec lines describe details of how the data is stored. Note that this is independent of the structure of the schema tree. A given schema tree can map back to data that is stored in very different ways, like a flat file, a web page, a relational database or an object oriented database, but share the same logical structure, by the use of different annotations for the input and output.

Detailed Description Text (218):
Multiple instances of a data type, such as multiple rows of a database table, in which each row has the same format, are specified by using the extended BNF plus ("+") operator to indicate "one or more occurrences" of that node. Manifest 112, element 140, and node 150 which appear in FIG. 3 are three examples of plus nodes. The number of instances of a "+" node can be optionally specified, either as a fixed constant or arithmetic expression, or it can be specified by a variable in the HLDSS file, such as element and node in the previous example. If no bound is specified, then the node type is recursive.

Detailed Description Text (224):
A uniform region is essentially a contiguous set of data from a common data source such that it is uniformly parsable, e.g., tuples from database relations, or binary data from a file.

Detailed Description Text (230):
An individual or atomic field, relative to its data source, may have further decomposition specified for it by the HLDSS. As an example, in a relational database, if a field is a binary large object, then it would appear as a single atomic field in the relational tuple, but it may have further substructure and thus would be decomposable by additional parsing. This would be indicated by an explicit annotation on the production that defines the substructure of that field, indicating the substructure to be applied--this may be text parsing of a comment, or binary data parsing of graphic or video data, for example. This would not change the nature of the containing uniform region (i.e., as a relational database region) since the data comes from the same data source and the substructure is of a single data component or field value.

Detailed Description Text (232):
Parsers can be organized as object types according to the taxonomy of data resource types. An instance of a parser object type is associated with a specific uniform region in the LSD schema. That parser instance is activated whenever data from that uniform region is to be accessed or created. Different uniform regions will have different instances of the same parser object type when these regions have the same type of data source and same type of data representation--e.g. objects in an object database. Utilizing different parser object instances, even of the same type, captures the possible differences in data sources (e.g., two different ASCII files) and the different states those parser object instances may be in when (an occurrence of) their region is complete.

Detailed Description Text (242):
This process is the same for all parser controller nodes, except that the top node of the LSD returns its instance tree for subsequent processing. If the LSD schema has cycles, each cycle is treated as recursive invocation of the subgraph pattern as the basis for parsing, until the data is exhausted. In principle, a uniform region could be as small as a single terminal node, or as large as the whole LSD.

Detailed Description Text (244):
The operation of specific types of uniform region data access functions, referred to as parsers will now be addressed. All relational database parsers share substantial functionality, and this is accomplished by the definition of a relational parser object type. Then specific relational databases are handled by creating associated parser object subtypes, such as for Oracle, Sybase, and Informix database products.

Detailed Description Text (245):
The methods associated with the relational parser object type include: 1) connecting to the database server, 2) opening the database, 3) issuing the query and opening the associated virtual or actual relation (the result of a query which joins relations is a virtual relation that is materialized by the database system), 4) iteratively accessing (retrieving or inserting/updating) each individual tuple and advancing the database cursor, 5) committing the transaction (for insert/update) to end the access, and 6) closing the database and disconnecting from the server. Steps 3 and 4 may be repeated for each of several queries before the subsequent completion steps are initiated. For some system interfaces, connecting and opening may be a single combined operation--in this case, the specialized object subtype for this interface would have a null method for step 1, while the method for step 2 would perform the combined operation.

Detailed Description Text (246):

The relational parser object type initiates a method to open the database (step 2) when the uniform region is about to be processed. If the connection has not been made to the database server by a previous uniform region, then a method for step 1 is executed. Following issuance of the query, individual tuples are retrieved--this is actually step 4a, as step 4 consists of two parts. Step 4b iterates over each attribute field of a retrieved tuple to create another terminal data instance node for the instance tree of this uniform region. When the fields of a tuple have been processed, then step 4a is repeated to obtain the next tuple, until all tuples have been processed.

Detailed Description Text (247):
After all tuples for a query have been processed, or that query is otherwise exited, step 5 commits the transaction, thereby allowing the database to release access to the underlying tables that serviced this query--typically this is needed only for multi-step insert and update access but not for simple query retrieval. Step 6 to close the database is executed only when the system has determined that no further access to this database is needed, or at the end of processing the LSD and all data for it.

Detailed Description Text (248):
These generic relational database operations are further specialized for Oracle, Sybase, and Informix parsers, and for other relational systems, by specializing each of the above methods for the specific syntax and operations required by the DBMS.

Detailed Description Text (249):
Both relational and object database parser types are themselves subtypes of the `database` object type in the system. This serves to abstract from the above methods those operations which are in common among all databases, such as generic initiation of a connection and opening the database, establishing a query or access pattern, accessing data instances, completing the access, and eventually disconnecting from the database.

Detailed Description Text (250):
For object databases, query-based access can be used to establish a focus within the OODB. Then traversal of the object instances leads to the particular data instances of interest. Each OODB system has its own query language--and some have no query language so that only navigational access is possible. In general, OODB query languages are moving toward some form of object-SQL.

Detailed Description Text (252):
After the query is executed and the objects have been accessed, the connection to the database may be closed when the system determines that no other access to that database will be needed or upon completion of all interactions.

Detailed Description Text (253):
Structured files, databases, and other data resources are treated uniformly--the same HLDSS specification language and LSD internal representation applies to all heterogeneous representations. The differences are dictated by the annotations.

Detailed Description Text (255):
For an ASCII region, data may be delimited (by whitespace or specified delimiters) or may consist of a specified number of characters, where this length may be determined by the value of other data which precedes the variable length string. Numeric ASCII data may be represented in standard base ten notation or, for integers, in hexadecimal or octal notation (hex begins with "0x" and octal has a leading zero). The number of elements in a sequence or repeating group of elements may be determined from the value of prior data, by encountering a different data type, or else by finding that the next data characters match a specified pattern (regular expression)--the alternative is specified by the HLDSS.

Detailed Description Text (258):
Application Programming Interfaces (APIs) consist of one or more procedures or methods that serve as the interface to a subsystem or a software package. Relational database access, discussed above, is an example of an API, though it is treated separately because of its special nature. Other API interfaces include HDF and netCDF data files, which are best accessed through the software interfaces provided for them. These API interfaces are characterized by one or more open/connect procedures, procedure invocation to specify relevant data, one or more data fetch operations (e.g., all the data at once or piece at a time), completion and closing procedures. The details of each API interface need to be implemented by program fragments, though some higher level specification is possible. The program fragments which may he needed should be relatively small and confined in scope, as they would be concerned with the local details of the API--the parsing and decomposition of the resulting data would be handled by the system based upon the HLDSS.

Detailed Description Text (264):
Several user interfaces may be used. All are subordinate to and accessible from the main interface which has been nicknamed DAISy, for DAtabase Integration System This is an X windows interface and it represents the Interoperability Assistant and Toolkit. It provides a great deal of support for building and executing information bridges along with other functionality. Consequently, it is intended to be used by both the Integration Administrator as well as by general users of information bridges.

Detailed Description Text (274):
An interactive browser has been developed for viewing and traversing both the schema and the data instances arising from the heterogeneous databases and design files. Unlike the external data viewers above, with the browser of the present invention the representation of data it presents is uniform regardless of the kinds of data source(s) involved. This representation is based on logical structure diagram (LSD) formalism described earlier.

Detailed Description Text (275):
The browser provides a homogeneous presentation of both the schema and data instances from multiple heterogeneous databases and structured design files. The represents an important aspect of integration since the user need not be concerned with the differences between data models, databases, and other structured data representations. With this feature in an information bridge, the ability to graphically represent data in a uniform manner without relying on particular data models is provided.

Detailed Description Text (279):
The Browser example in FIG. 5 shows that the data values at the leaves of the schema are displayed. Data is associated only with the leaves, while the tree structure represents relationships in the source and/or target databases. Notice that only one set of related data values is shown at a time--that is, one data value for each leaf node in the LSD schema tree.

Detailed Description Text (301):
A semantic data specification language (SEMDAL) incorporates features for management of metadata for databases, structured data, document structure, and collections of multimedia documents as well as web-based information. Metadata is descriptive information about data instances. This metadata may provide the schema structure in which the instance data is stored and/or presented. The metadata also may provide a wide variety of other information to help interpret, understand, process, and/or utilize the data instances. At times, metadata may be treated as if it were instance data, depending upon the application.

Detailed Description Text (302):
Both Standard Generalized Markup Language (SGML) and the high level data structure (HLDSS) of the present invention are subsumed by SEMDAL. SEMDAL includes frame based structures, inheritance, creation of groups of frames, etc. A subset of SEMDAL can be transformed into the HLDSS format. This enables SEMDAL to be applied to database interoperability issues for structured databases (relational, object-oriented, hierarchical, etc.) as well as for structured files.

Detailed Description Text (303):
While a subset of SEMDAL translates into the HLDSS, a somewhat different but significantly overlapping subset translates into SGML and thus offers a different surface syntax. Thus, it is the interpretation associated with the syntax that is essential. Normally, SGML is interpreted to refer to document structuring with tags (markup) and HLDSS refers to databases and structured data. However, with SEMDAL different interpretations may be associated to a SEMDAL construct to achieve either of these results. This facilitates bridging the gap between structured databases and free form text and documents.

Detailed Description Text (311):
If, however, INTERPretation is specified as a `relational database`, then in the previous example Frame, manifest is a relation which has fields/attributes for header, bound, nodes and numNodes. In fact, what is referred to as the `phyla` in the HLDSS for database interoperability is essentially what is meant here by the interpretation. This is what determines whether the HLDSS statement is to be treated as a reference to a relation, an object, or part of a parsing specification for a structured file.

Detailed Description Text (313):
The notion of interpretation enables the use of the same representation to express a matrix. This is one of the novel aspects of how interpretation is used in the SEMDAL language to unify and express seemingly diverse constructs such as structured documents, relational and object-oriented databases, and matrices.

Detailed Description Text (317):
In view of the above, matrices can be represented in a relational database by a relation defined for each matrix and having an attribute/field for each of the N dimensions, and another attribute for the value--or several attributes if the value has substructure. Then each tuple would contain the N coordinates followed by the value at that coordinate. This could be space efficient for sparse matrices.

Detailed Description Text (323):
Note that the above INTERPretation as a `relation` is not the same as a relational database, since here it has not been committed how this 4-ary Supplies relation is to be materialized and stored. This relationship could be, if one wanted, represented by Horn logic clauses in a prolog system.

Detailed Description Text (337):
3. Constrains express the requirements for data consistency within complex databases and between different sites that contain interrelated data values.

Detailed Description Text (340):
6. Constraints go beyond the trigger mechanisms that have been and are being introduced into commercial database systems.

Detailed Description Text (354):
The translation of SEMDAL to a logic language such as KIF (Knowledge Interchange Format) has yet to be developed for reasoning and consistency analysis of a set of semantic constraints. The notion of constraint packages with enforcement actions to maintain data consistency among distributed heterogeneous databases also needs to be further developed.

Detailed Description Text (356):
The logical design level of the metadata repository allows for multiple implementations and is convenient in different storage architectures. The physical storage of one implementation is relation-based in order to be accessible with the Java Database Connectivity (JDBC) standard for the Java interfaces to relational databases. Note that the choice of repository implementation is independent from the data models being described.

Detailed Description Text (363):
At the structural level, the primary characteristics of SGML (Standard Generalized Markup Language), HTML (Hypertext Markup Language), and the new evolving XML (Extensible Markup Language) have been subsumed, as well as the heterogeneous database structures--including relational and object-oriented models, and to provide extensibility to address multimedia data.

Detailed Description Text (364):
Although the existing representations for structured documents, databases, and semantic knowledge representation are rather different on the surface, unification at the logical level has been achieved. The common framework, then, admits alternative syntactic presentations--the need for which has heretofore accounted for the superficially large differences between different specification languages.

Detailed Description Text (366):
It is tempting to refer to this logical representation as the `internal` representation, but in fact this single logical representation could have its physical storage implementation in a relational database, or an object-database, or in a hierarchical or network database architecture just as easily. A relational storage implementation has been chosen because of its accessibility through the increasingly popular JDBC application programming interface.

Detailed Description Text (380):
This features of the logical model can be represented and stored in an object-database or a nested-relational database (there are few nested relational databases). It can be conveniently represented in a traditional relational database through the use of keys which reference the subordinate hierarchical levels. Similarly, the structural hierarchy is reflected by tuples in a different relation, with similar ability to find the parent aggregate and/or the component children of a non-terminal node. Thus each level of the semantic hierarchy and the structural hierarchy are represented by tuples in relational tables, and references to subordinate (or parent) levels is effected through the relational join operation.

Detailed Description Text (384):
The representation for an analog watch then can refer to Clock-Schema.SecondHand.Length.Units to define the units in which the length of the second hand will be expressed. Then a simple number in the actual data representation will have meaning that is explicit. Such units is semantic information and is part of the metadata that is managed, and it is necessary information in order to properly utilize the actual length data values--which may be stored in a separate database of instance data.

Detailed Description Text (397):
Each of these are referred to as different interpretations of the same syntactic representation. SEMDAL provides a system level attribute `INTERP` to capture this INTERPretation, indicating, for example, whether the data is from a relational database table or from the data members of an object.

Detailed Description Text (420):
In the following example, both an object database (ODB) and a relational database (RDB) as well as a photo are represented. The name of the MetaFrame is MDS1--often

the name of the frame may be given to be the same as the first leftmost component (here `Manifest`). Note that "::" is an abbreviation for substructure.

Detailed Description Text (421):
The ODB above has substructure consisting of a set of header information--the "+" indicates one or more occurrences of this information, similarly for bound. Thus the object database contains a set of bound information, while the relational database contains a table with all the header information.

Detailed Description Text (425):
The logical structure described for the Metadata repository is hierarchical, which could be naturally implemented in a `nested relational` database or an object database, as well as in other databases. Due to the increasing value of the JDBC connectivity between Java programs and relational databases, a relational implementation has been chosen. Thus the Metadata Repository will be accessible via the JDBC standard API. The metadata which are stored describe related instance data--which usually is stored separately in one or more repositories, such as in a relational or object database, document repository , or multimedia digital library.

Detailed Description Text (436):
The substructure portion of a MetaFrame consists of those lines, typically placed first in the MetaFrame, which have "::" following a token--or an attribute path ending in substructure--these are often referred to as structure productions. To represent this information in a relational database, to create a relational table MetaFrameStructure:

Detailed Description Text (440):
Note that all RHS's which do not appear as a LHS are `terminals` in the sense of the substructure `grammar`. All RHS's which do appear as a LHS are `non-terminals` and do not generally correspond to individual data values in the database being described, but rather are placeholders for the substructure.

Detailed Description Text (441):
A structure path expression may begin with a FrameName and consists of dot-separated terms, each term being a LHS or RHS, such that the RHS must correspond to that LHS. If this LHS also appears as a RHS, the structure path may continue with a corresponding LHS. Thus in the MDS1 example on page 8, the structure path "Manifest.bound.scaleX" could be used to refer to this structure, the associated value, as well as any other semantic attributes of scaleX. Also, a structure path expression ending in a non-terminal can be used to refer to the data structure instance(s) corresponding to this non-terminal--each instance itself being a data structure consisting of the multiple data values and their associated structure, as described by the MetaFrame productions. Structure path expressions can be used in a generic uniform query language to refer to data in any form of structure or database, based upon the metadata in the repository.

Detailed Description Text (442):
In some systems, such as relational databases, the basic table definitions and some related information are stored in `system catalogs` which themselves are relational tables which may be accessed by the user in read-only mode. All such data may be accommodated in the structure as shown so far. An example of this kind of catalog information, taken from Informix, is:

Detailed Description Text (454):
Sometimes the descriptive metadata is of the same nature for each entry. For example, the metadata for a relational database usually will consist of the same kinds of descriptive attributes for each relational table and for each column, just with different values. In these cases a more compact representation as a table of Metadata values is possible--though doing imparts no additional information. That

is, if the kinds of semantic attributes are the same for each, the same attribute subpath does not need to be repeated for each different table and value. Rather, each attribute subpath could be taken as a column name in a meta-table to provide greater conciseness.

Detailed Description Text (462):
The DAtabase Integration System DAISy) of the present invention has been described in detail and how it provides interoperability across heterogeneous resources, including databases and other forms of structured data. In this system, the major differences between diverse data representations are accommodated by high level LSD (Logical Structure Diagram) specification language, and by use of annotations to factor out of this declarative language the heterogeneity among different databases and data structure representations.

Previous Doc Next Doc Go to Doc#

L14: Entry 31 of 33 File: USPT Oct 19, 1999

US-PAT-NO: 5970490
DOCUMENT-IDENTIFIER: US 5970490 A

TITLE: Integration platform for heterogeneous databases

DATE-ISSUED: October 19, 1999

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Morgenstern; Matthew	Ithaca	NY		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Xerox Corporation	Stamford	CT			02

APPL-NO: 08/963853 [PALM]
DATE FILED: November 4, 1997

PARENT-CASE:
This application claims priority of Provisional U.S Pat. Application No. 60/030,215, filed Nov. 5, 1996 the subject matter of this application is fully incorporated herein.

INT-CL-ISSUED: [06] G06 F 17/30

US-CL-ISSUED: 707/10; 707/103, 707/104
US-CL-CURRENT: 707/10; 707/104.1

FIELD-OF-CLASSIFICATION-SEARCH: 707/10, 707/103, 707/104
See application file for complete search history.

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

Search Selected Search ALL Clear

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
5560005	September 1996	Hoover et al.	707/10
5627979	May 1997	Chang et al.	345/335
5724575	March 1998	Hoover et al.	707/10
5758351	May 1998	Gibson et al.	707/104
5761684	June 1998	Gibson	707/515

☐	5809507	September 1998	Cavanaugh, III	707/103
☐	5815415	September 1998	Bentley et al.	364/578

OTHER PUBLICATIONS

Common Object Request Broker Architecture, http://www.sei.cmu.edu/activities/str/descriptions/corba.sub.- body.html, Jan. 10, 1997.
Object Request Broker, http://www.sei.cmu.edu/activities/str/descriptions/orb.sub.-body.html. Jun. 25, 1997.

ART-UNIT: 277

PRIMARY-EXAMINER: Amsbury; Wayne

ASSISTANT-EXAMINER: Alam; Shahid

ATTY-AGENT-FIRM: Cox; Diana M.

ABSTRACT:

A method for processing heterogeneous data including high level specifications to drive program generation of information mediators, inclusion of structured file formats (also referred to as data interface languages) in a uniform manner with heterogeneous database schema, development of a uniform data description language across a wide range of data schemas and structured formats, and use of annotations to separate out from such specifications the heterogeneity and differences that heretofore have led to costly special purpose interfaces with emphasis on self-description of information mediators and other software modules.

18 Claims, 5 Drawing figures

Previous Doc Next Doc Go to Doc#